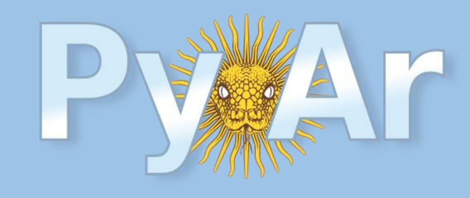
2016

Python

APRENDIENDO A PROGRAMAR









Federico N. Brest

federiconbrest@gmail.com

Contenido

Prefacio

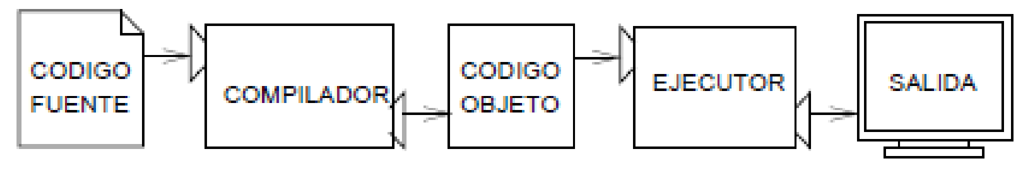
Este manual pretende introducir a cualquier persona, que haya programado en cualquier otro lenguaje o no, en el desarrollo de software con **Python**. Solo es una iniciación, por lo que se dejan sólo una simple descripción de algunos conceptos, los cuales pueden ser profundizados por el alumno por su propia cuenta.
Inspirado en un nido de horas estudiando y programando en **Python**, este texto está redactado para ser leído (y probado) de manera secuencial, introduciendo progresivamente los conceptos fundamentales del lenguaje en los diversos temas.
Puesto que no es lo mismo andar el camino que conocer el camino, se insta al lector a que pruebe todos los ejemplos que vienen en el documento. Y es que para aprender a desarrollar “en cualquier lenguaje de programación” hay que embarrarse las manos, no hay otra forma.
Con este material espero que se vea atraído por este lenguaje y disfrute tanto como nosotros aplicar horas al desarrollo de código bajo esta herramienta.

¿Qué es Python?

Python es un lenguaje de programación creado por Guido van Rossum a principios de los años 90 cuyo nombre está inspirado en el grupo de cómicos ingleses “Monty Python”.
Se trata de un lenguaje interpretado o de script, con tipado dinámico, fuertemente tipado, multiplataforma y orientado a objetos.
Tipado dinámico: La característica de tipado dinámico se refiere a que no es necesario declarar el tipo de dato que va a contener una determinada variable, sino que su tipo se determinará en tiempo de ejecución según el tipo del valor al que se asigne, y el tipo de esta variable puede cambiar si se le asigna un valor de otro tipo.
Fuertemente tipado: No se permite tratar a una variable como si fuera de un tipo distinto al que tiene, es necesario convertir de forma explícita dicha variable al nuevo tipo previamente. Por ejemplo, si tenemos una variable que contiene un texto (variable de tipo cadena o string) no podremos tratarla como un número (sumar la cadena “9” y el número 8). En otros lenguajes el tipo de la variable cambiaría para adaptarse al comportamiento esperado, aunque esto es más propenso a errores.
Python es un ejemplar de un lenguaje de alto nivel; otros ejemplos de lenguajes de alto nivel son C, C++, Perl y Java.
Como se puede deducir de la nomenclatura “lenguaje de alto nivel”, también existen lenguajes de bajo nivel, a los que también se califica como lenguajes de máquina o lenguajes ensambladores. A propósito, los computadores sólo ejecutan programas escritos en lenguajes de bajo nivel. Los programas de alto nivel tienen que traducirse antes de ejecutarse. Esta traducción lleva tiempo, lo cual es una pequeña desventaja de los lenguajes de alto nivel.
Aun así las ventajas son enormes. En primer lugar, la programación en lenguajes de alto nivel es mucho más fácil; escribir programas en un lenguaje de alto nivel toma menos tiempo, los programas son más cortos y más fáciles de leer, y es más probable que estos programas sean correctos. En segundo lugar, los lenguajes de alto nivel son portables, lo que significa que pueden ejecutarse en tipos diferentes de computadores sin modificación alguna o con pocas modificaciones.
Los programas escritos en lenguajes de bajo nivel sólo pueden ser ejecutarse en un tipo de computador y deben reescribirse para ejecutarlos en otro.
Debido a estas ventajas, casi todos los programas se escriben en un lenguaje de alto nivel. Los lenguajes de bajo nivel sólo se usan para unas pocas aplicaciones especiales.
Hay dos tipos de programas que traducen lenguajes de alto nivel a lenguajes de bajo nivel: intérpretes y compiladores. Un **intérprete** lee un programa de alto nivel y lo ejecuta, lo que significa que lleva a cabo lo que indica el programa.
Traduce el programa poco a poco, leyendo y ejecutando cada comando.



Un **compilador** lee el programa y lo traduce todo al mismo tiempo, antes de ejecutar cualquiera de las instrucciones. En este caso, al programa de alto nivel se le llama el **código fuente**, y al programa traducido el **código de objeto** o el **código ejecutable**. Una vez compilado el programa, puede ejecutarlo repetidamente sin volver a traducirlo.



Python se considera como lenguaje interpretado porque los programas de Python se ejecutan por medio de un intérprete.

¿Por qué Python?

Python es un lenguaje que todo el mundo debería conocer. Su sintaxis simple, clara y sencilla; el tipado dinámico, el gestor de memoria, la gran cantidad de librerías disponibles y la potencia del lenguaje, entre otros, hacen que desarrollar una aplicación en Python sea sencillo, muy rápido y, lo que es más importante, divertido.
La sintaxis de Python es tan sencilla y cercana al lenguaje natural que los programas elaborados en Python parecen pseudocódigo. Por este motivo se trata además de uno de los mejores lenguajes para comenzar a programar.
Python no es adecuado sin embargo para la programación de bajo nivel o para aplicaciones en las que el rendimiento sea crítico.
Algunos casos de éxito en el uso de Python son Google, Yahoo, la NASA, y todas las distribuciones Linux, en las que Python cada vez representa un tanto por ciento mayor de los programas disponibles.

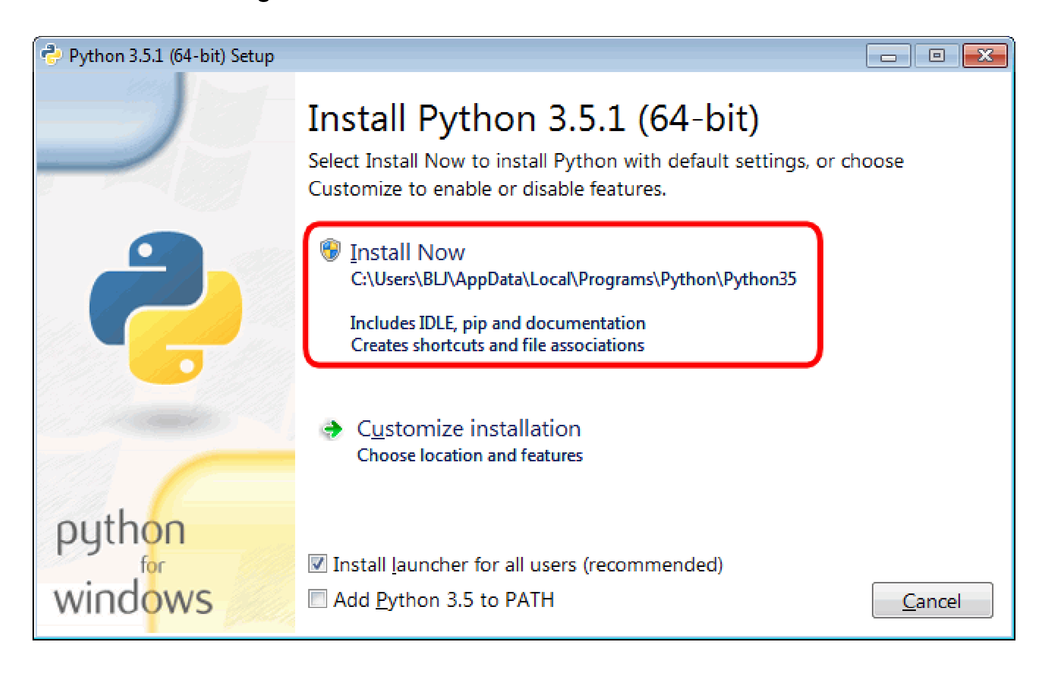
Instalación de Python

En Windows

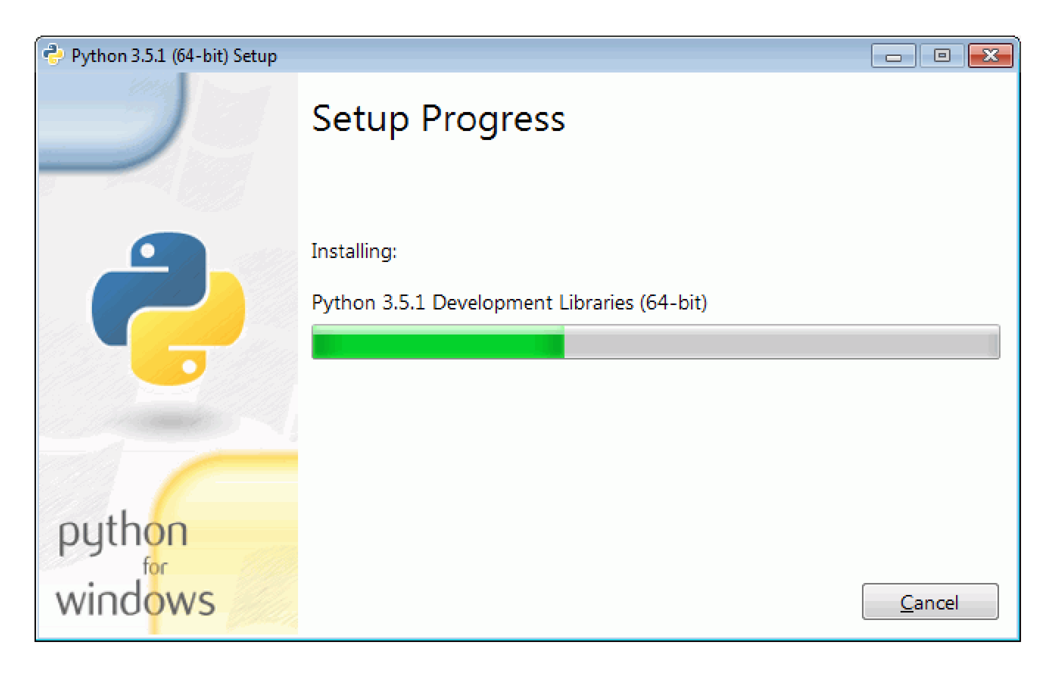
La página oficial de Python es https://www.python.org/.
La última versión de Python 3.5 disponible actualmente (enero de 2016) es Python 3.5.1 (del 7 de diciembre de 2015).
Una vez descargado el instalador, haciendo doble clic en él se inicia la instalación.

Nota: Las imágenes siguientes corresponden a la instalación de Python 3.5.1 (64 bits), pero son similares en cualquier versión 3.5.X.

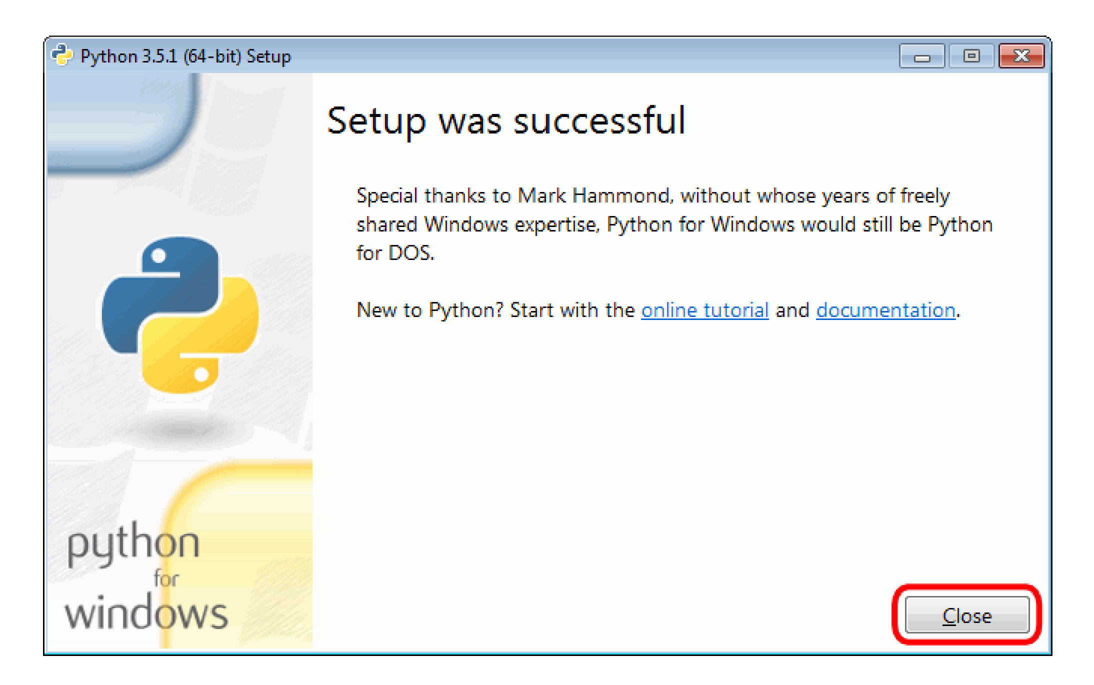
La primera pantalla permite seleccionar las opciones de instalación. En principio, no es necesario modificarlas. Para continuar, haga clic en el botón "Install Now".



A continuación se iniciará el proceso de instalación. La instalación de Python puede durar varios minutos.



Una vez completada la instalación, se mostrará la pantalla final. Haciendo clic en en el botón "Close" se cerrará el programa de instalación.
Una vez terminada la instalación, ¡ya puede empezar a programar en Python!



En Linux

Si ya tiene instalado GNU/Linux en tu PC, tendrá Python instalado en su sistema.
Para comprobarlo, abra una terminal y escriba python como se muestra a continuación:

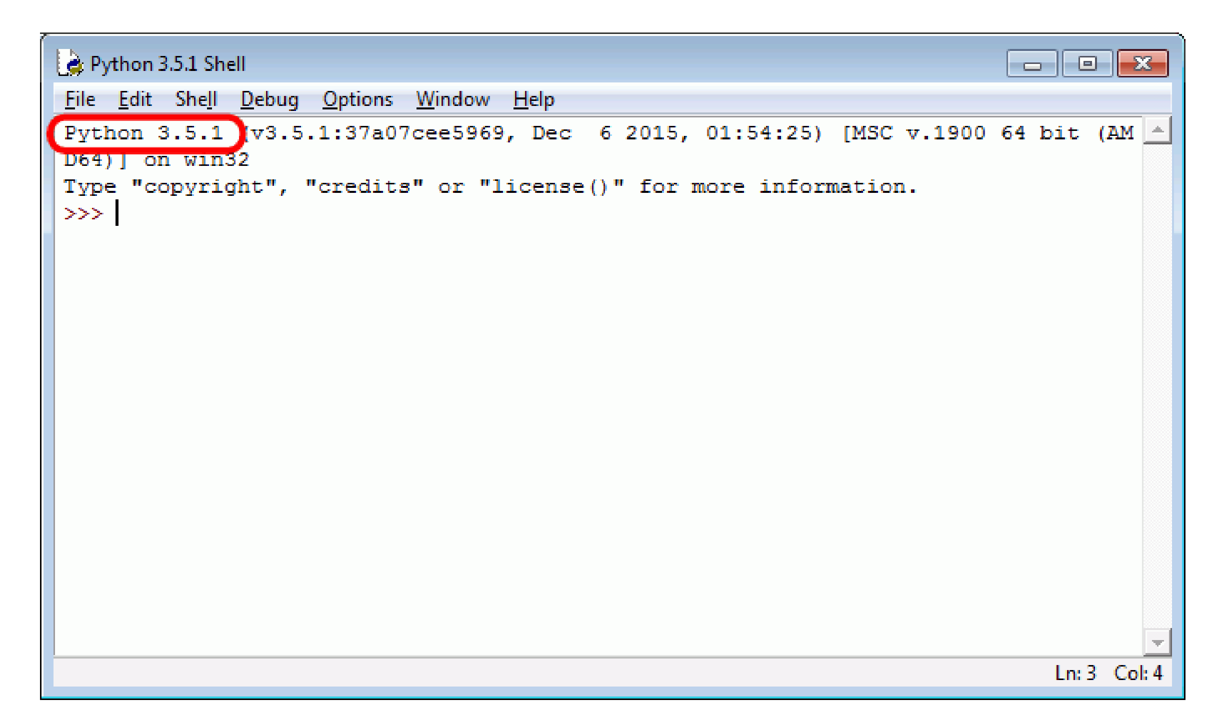
```
fede@Fede-Notebook:~$ python
Python 2.7.11+ (default, Apr 17 2016, 14:00:29)
[GCC 5.3.1 20160413] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

Lo que verá en pantalla, es el Shell interactivo de Python.
La primera línea nos indica la versión de Python que tenemos instalada. Al final podemos ver el prompt (>>>) que nos indica que el intérprete está esperando código del usuario.
Para salir del Shell interactivo, pulse las teclas Ctrl + D.
Si en lugar del Shell interactivo, ves un mensaje de error similar a "python: orden no encontrada", deberá seguir los siguientes pasos para instalarlo:

1. Actualice la lista de los repositorios: `sudo apt-get update`
2. Actualice el Sistema Operativo: `sudo apt-get upgrade`
3. Instale Python: `sudo apt-get install python2.7`

Ejecutar IDLE

Python es el nombre del lenguaje de programación.
IDLE (Integrated DeveLopment Environment for Python) es el entorno de desarrollo que permite editar y ejecutar los programas. Se pueden crear y ejecutar programas en Python sin utilizar IDLE, pero IDLE hace mucho más fáciles esas tareas (de hecho, IDLE en inglés significa holgazán).
Al abrir IDLE con el acceso directo Inicio > Todos los programas > Python 3.5 > IDLE (Python 3.5), se abrirá la ventana principal de IDLE, como muestra la imagen siguiente.



Esta ventana indica la versión de Python instalada.

Colores en los programas

Al escribir órdenes en IDLE, algunas palabras cambian de color. Los colores ayudan a identificar los distintos tipos de elementos y a localizar errores:

- Las palabras reservadas de Python (las que forman parte del lenguaje) se muestran en color naranja.
- Las cadenas de texto se muestran en verde.
- Los resultados de las órdenes se escriben en azul.
- Los mensajes de error se muestran en rojo.
- Las funciones se muestran en púrpura.

Mi primer programa

Como comentábamos en el capítulo anterior existen dos formas de ejecutar código Python, bien en una sesión interactiva (línea a línea) con el intérprete, o bien de la forma habitual, escribiendo el código en un archivo de código fuente y ejecutándolo.
El primer programa que vamos a escribir en Python es el clásico Hola Mundo, y en este lenguaje es tan simple como:

```
print “Hola Mundo”
```

Vamos a probarlo primero en el intérprete. Ejecute Python desde consol.a, escriba la línea anterior y pulsa Enter. El intérprete responderá mostrando en la consola el texto `Hola Mundo`.
Vamos ahora a crear un archivo de texto con el código anterior, de forma que pudiéramos distribuir nuestro pequeño gran programa entre nuestros amigos. Abre tu editor de texto preferido o bien el IDE que hayas elegido y copia la línea anterior. Guárdalo como hola.py, por ejemplo.
Ejecutar este programa es tan sencillo como indicarle el nombre del archivo a ejecutar al intérprete de Python

```
python hola.py
```

Comentarios

Un archivo, no solo puede contener código fuente. También puede incluir comentarios (notas que como programadores, indicamos en el código para poder comprenderlo mejor).
Los comentarios pueden ser de dos tipos: de una sola línea o multi-línea y se expresan de la siguiente manera:

```
# Esto es un comentario de una sola línea
mi_variable = 15
"""Y este es un comentario
de varias líneas"""
mi_variable = 15
mi_variable = 15 # Este comentario es de una línea
también
```

Usar Python como una calculadora

Vamos a probar algunos comandos simples en Python. Iniciá un intérprete y esperá por el prompt primario, >>>. (No debería demorar tanto).

Números

El intérprete actúa como una simple calculadora; puede ingresar una expresión y este escribirá los valores. La sintaxis es sencilla: los operadores +, -, * y / funcionan como en la mayoría de los lenguajes (por ejemplo, Pascal o C); los paréntesis (()) pueden ser usados para agrupar. Por ejemplo:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5 # la división siempre retorna un número de
punto flotante
1.6
```

Los números enteros (por ejemplo 2, 4, 20) son de tipo `int`, aquellos con una parte fraccional (por ejemplo 5.0, 1.6) son de tipo `float`. Vamos a ver más sobre tipos de números luego. La división (/) siempre retorna un punto flotante. Para hacer *floor division* y obtener un resultado entero (descartando cualquier resultado fraccional) puede usar el operador //; para calcular el resto puede usar %:

```
>>> 17 / 3  # la división clásica retorna un punto
flotante
5.666666666666667
>>> 17 // 3  # la división entera descarta la parte
fraccional
5
>>> 17 % 3  # la división entera descarta la parte
fraccional
2
```

Con Python, es posible usar el operador ** para calcular potencias

```
>>> 5 ** 2  # 5 al cuadrado
25
>>> 2 ** 7  # 2 a la potencia de 7
128
```

Variables

Una variable es un espacio para almacenar datos modificables, en la memoria de un ordenador. En Python, una variable se define con la sintaxis:

```
nombre_de_la_variable = valor_de_la_variable
```

Cada variable, tiene un nombre y un valor, el cual define a la vez, el tipo de datos de la variable.
Una variable puede almacenar números, texto o estructuras más complicadas (que se verán más adelante). Si se va a almacenar texto, el texto debe escribirse entre comillas simples (') o dobles ("), que son equivalentes. A las variables que almacenan texto se les suele llamar cadenas (de texto).
El nombre de una variable debe empezar por una letra o por un guión bajo (_) y puede seguir con más letras, números o guiones bajos. Pueden contener mayúsculas, pero tenga en cuenta que Python distingue entre mayúsculas y minúsculas (en inglés se dice que Python es "case-sensitive").
El signo igual (=) es usado para asignar un valor a una variable. Luego, ningún resultado es mostrado antes del próximo prompt:

```
>>> ancho = 20
>>> largo = 5 * 9
>>> ancho * largo
900
```

Si una variable no está "definida" (con un valor asignado), intentar usarla producirá un error:

```
>>> n  # tratamos de acceder a una variable no
definida
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Además de `int` y `float`, Python soporta otros tipos de números, como ser `Decimal` y `Fraction`. Python también tiene soporte integrado para númreos complejos, y usa el sufijo j o J para indicar la parte imaginaria (por ejemplo 3+5j).

Palabras reservadas (keywords)

Las palabras reservadas de Python son las que forman el núcleo del lenguaje Python. Son las siguientes:

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

Estas palabras no pueden utilizarse para nombrar otros elementos (variables, funciones, etc.), aunque pueden aparecer en cadenas de texto.

Operadores

Los operadores son los caracteres que definen operaciones matemáticas (lógicas y aritméticas). Son los siguientes:

Operador	Descripción	Ejemplo
+	Suma	r = 3 + 2 # r es 5
-	Resta	r = 4 - 7 # r es -3
-	Negación	r = -7 # r es -7
*	Multiplicación	r = 2 * 6 # r es 12
**	Exponente	r = 2 ** 6 # r es 64
/	División	r = 3.5 / 2 # r es 1.75
//	División entera	r = 3.5 // 2 # r es 1.0
%	Módulo	r = 7 % 2 # r es 1

Cadenas de caracteres

Además de números, Python puede manipular cadenas de texto, las cuales pueden ser expresadas de distintas formas. Pueden estar encerradas en comillas simples ('...') o dobles ("...") con el mismo resultado \ puede ser usado para escapar comillas:

```
>>> 'huevos y pan' # comillas simples
 'huevos y pan'
>>> 'doesn\'t' # usa \' para escapar comillas simples...
"doesn't"
>>> "doesn't" # ...o de lo contrario usa comillas doblas
"doesn't"
>>> ' "Si," le dijo.'
' "Si," le dijo.'
>>> "\"Si,\" le dijo."
'"Si," le dijo.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

En el intéprete interactivo, la salida de cadenas está encerrada en comillas y los caracteres especiales son escapados con barras invertidas. Aunque esto a veces luzca diferente de la entrada (las comillas que encierran pueden cambiar), las dos cadenas son equivalentes. La cadena se encierra en comillas dobles si la cadena contiene una comilla simple y ninguna doble, de lo contrario es encerrada en comillas simples. La función `print()` produce una salida más legible, omitiendo las comillas que la encierran e imprimiendo caracteres especiales y escapados:

```
>>>' "Isn\'t," she said.'
 '"Isn\'t," she said.'
>>> print('"Isn\'t," she said.')
"Isn't," she said.
>>>s = 'Primerea línea.\nSegunda línea.' # \n significa
nueva línea
>>>s # sin print(), \n es incluído en la salida
'Primera línea.\nSegunda línea.'
>>> print(s) # con print(), \n produce una nueva línea
Primera línea.
Segunda línea.
```

Si no quiere que los caracteres antepuestos por \ sean interpretados como caracteres especiales, puede usar cadenas crudas agregando una r antes de la primera comilla:

```
>>> print('C:\algun\nombre')  # aquí \n significa nueva
línea!
C:\algun
ombre
>>> print(r'C:\algun\nombre')  # note la r antes de la
comilla
C: \algun\nombre
```

Las cadenas de texto pueden ser concatenadas (pegadas juntas) con el operador + y repetidas con *:

```
>>> # 3 veces 'un' seguido de 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Dos o más cadenas literales (aquellas encerradas entre comillas) una al lado de la otra son automáticamente concatenadas:

```
>>> 'Py' + 'thon'
'Python'
```

Esto solo funciona con dos literales, no con variables ni expresiones.
Si se quiere concatenar variables o una variable con un literal, se debe usar +:

```
>>> prefix = 'Py'
>>> prefix + 'thon'
'Python'
```

Las cadenas de texto se pueden *indexar* (subíndices), el primer carácter de la cadena tiene el índice 0. No hay un tipo de dato para los caracteres; un carácter es simplemente una cadena de longitud uno:

```
>>> palabra = 'Python'
>>> palabra[0]   # caracter en la posición 0
'P'
>>> palabra[5]   # caracter en la posición 5
'n'
```

Además de los índices, las rebanadas también están soportadas. Mientras que los índices son usados para obtener caracteres individuales, las rebanadas te permiten obtener sub-cadenas:

```
>>> palabra = 'Python'
>>> palabra[0:2]  # caracteres desde la posición 0
(incluída) hasta la 2 (excluída)
'Py'
>>> palabra[2:5]  # caracteres desde la posición 2
(incluída) hasta la 5 (excluída)
'tho'
```

La función incorporada `len()` devuelve la longitud de una cadena de texto:

```
>>> s = 'supercalifrastilisticoespialidoso'
>>>len(s)
33
```

Listas

Python tiene varios tipos de datos *compuestos*, usados para agrupar otros valores. El más versátil es la ***lista***, la cual puede ser escrita como una lista de valores separados por coma (ítems) entre corchetes. En Python, una lista es un conjunto ordenado de elementos.
Una variable puede almacenar una lista completa y la variable hace referencia a la lista completa.

Una lista que no contiene ningún elemento se denomina **lista vacía**.

Las listas pueden contener ítems de diferentes tipos, pero usualmente los ítems son del mismo tipo:

```
>>> cuadrados = [1, 4, 9, 15, 25]
>>> cuadrados
[1, 4, 9, 15, 25]
```

Como las cadenas de caracteres (y todos los otros tipos *sequence* integrados), las listas pueden ser indexadas y rebanadas:

```
>>> cuadrados = [0]
[1]
>>> cuadrados = [-1]
[25]
>>> cuadrados = [-3 :]
[9, 16, 25]
```

Las listas también soportan operaciones como concatenación:

```
>>> cuadrados + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

En Python se distingue entre objetos mutables y objetos inmutables:

- Los objetos **inmutables** son objetos que no se pueden modificar. Por ejemplo, los números, las cadenas y las tuplas son objetos inmutables
- Los objetos **mutables** son objetos que se pueden modificar. Por ejemplo, las listas y los diccionarios son objetos mutables.

A diferencia de las cadenas de texto, que son *immutable*, las listas son un tipo *mutable*, es posible cambiar un su contenido:

```
>>> cubos = [1, 8, 27, 65, 125]  #Hay algo mal aqui
>>> 4 ** 3  #el cubo de 4 es 64, no 65!
64
>>> cubos[3] = 64  #reemplazar el valor incorrecto
>>> cubos
[1, 8, 27, 64, 125]
```

También se puede agregar nuevos ítems al final de la lista, usando el método `append()` (vamos a ver más sobre los métodos luego):

```
>>> cubos.append(216) #agregar el cubo de 6
>>> cubos.append(7 ** 3) #y el cubo de 7
>>> cubos
[1, 8, 27, 64, 125, 216. 343]
```

La función predefinida `len()` también sirve para las listas:

```
>>>letras = ['a', 'b', 'c', 'd']
>>>len(letras)
4
```

La palabra reservada del

La palabra reservada del permite eliminar un elemento o varios elementos a la vez de una lista, e incluso la misma lista.

```
>>> letras = ["A", "B", "C", "D", "E", "F", "G", "H"]
>>> del letras[4]   # Elimina la sublista ['E']
>>> letras
['A', 'B', 'C', 'D', 'F', 'G', 'H']
>>> del letras[1:4] # Elimina la sublista ['B', 'C', 'D']
>>> letras
['A', 'F', 'G', 'H']
```

El tipo range()

El tipo range() crea una lista inmutable de números enteros en sucesión aritmética. El tipo range() puede tener uno, dos o tres argumentos numéricos enteros.
El tipo range() con un único argumento se escribe range(n) y crea una lista creciente de *n* términos enteros que empieza en 0 y acaba antes de llegar a *n* (los términos aumentan de uno en uno). Es decir, range(n) = range(0, n) = [0, 1, ... , n-1].
Para ver los valores de la lista creada con range(), es necesario convertirla a lista mediante la función list().

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Si *n* no es positivo, se crea una lista vacía.
El tipo range() con dos argumentos se escribe range(m, n) y crea una lista creciente de términos enteros que empieza en m y acaba antes de llegar a n (los términos aumentan de uno en uno). Es decir, range(m, n) = [m, m+1, ... , n-1]

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
>>> list(range(-5, 1))
[-5, -4, -3, -2, -1, 0]
```

Si *n* es menor o igual que *m*, se crea una lista vacía.
El tipo range con tres argumentos se escribe range(m, n, p) y crea una lista que empieza en m y acaba antes de llegar a n, avanzando de pen p. El valor de p puede ser positivo o negativo. Es decir, range(m, n, p) = [m, m+p, ... , x] siendo x tal que n - p ≤ x < n (si p es negativo, n- p ≥ x > n)

```
>>> list(range(5, 21, 3))
[5, 8, 11, 14, 17, 20]
>>> list(range(10, 0, -2))
[10, 8, 6, 4, 2]
```

El valor de *p* no puede ser cero:
Si *p* es positivo y *n* menor o igual que *m*, o si *p* es negativo y *n* mayor o igual que *m*, range(m, n, p) devolverá una lista vacía.
En resumen, los tres argumentos del tipo range(m, n, p) son:

- *m*: el valor inicial
- *n*: el valor final (que no se alcanza nunca)
- *p*: el paso (la cantidad que se avanza cada vez).

Si se escriben sólo dos argumentos, Python le asigna a *p* el valor 1. Es decir range(m, n) es lo mismo que range(m, n, 1)
Si se escribe sólo un argumento, Python, le asigna a *m* el valor 0 y a *p* el valor 1. Es decir range(n) es lo mismo que range(0, n, 1).

Tuplas

Una tupla es una variable que permite almacenar varios datos inmutables (no pueden ser modificados una vez creados) de tipos diferentes:

```
mi_tupla = ('cadena de texto', 15, 2.8, 'otro dato', 25)
```

Se puede acceder a cada uno de los datos mediante su índice correspondiente, siendo 0 (cero), el índice del primer elemento:

```
print mi_tupla[1] # Salida: 15
```

También se puede acceder a una porción de la tupla, indicando (opcionalmente) desde el índice de inicio hasta el índice de fin:

```
print mi_tupla[1:4] # Devuelve: (15, 2.8, 'otro dato')
print mi_tupla[3:] # Devuelve: ('otro dato', 25)
print mi_tupla[:2] # Devuelve: ('cadena de texto', 15)
```

Otra forma de acceder a la tupla de forma inversa (de atrás hacia adelante), es colocando un índice negativo:

```
print mi_tupla[-1] # Salida: 25
print mi_tupla[-2] # Salida: otro dato
```

Salida por pantalla

En Informática, la "salida" de un programa son los datos que el programa proporciona al exterior. Aunque en los incios de la informática la salida más habitual era una impresora, hace muchos años que el dispositivo de salida más habitual es la pantalla del ordenador.
La forma más sencilla de mostrar algo en la salida estándar es mediante el uso de la sentencia print, como hemos visto multitud de veces en ejemplos anteriores. En su forma más básica a la palabra clave print le sigue una cadena, que se mostrará en la salida estándar al ejecutarse

La función print()

La función print() permite mostrar texto en pantalla. El texto a mostrar se escribe como argumento de la función:

```
print("Hola")
```

Las cadenas se pueden delimitar tanto por comillas dobles (") como por comillas simples (').

Nota: En estos apuntes se utilizan normalmente comillas dobles.

La función print() admite varios argumentos seguidos. En el programa, los argumentos deben separarse por comas. Los argumentos se muestran en el mismo orden y en la misma línea, separados por espacios:

```
print("Hola", "Adios")
```

Para generar una línea en blanco, se puede escribir una orden print() sin argumentos.
Si no se quiere que Python añada un salto de línea al final de un print(), se debe añadir al final el argumento end="".

Operando con datos introducidos por el usuario

Los programas que hemos escrito hasta ahora son un poco maleducados en el sentido de que no aceptan entradas de datos del usuario. Simplemente hacen lo mismo siempre.
En Informática, la "entrada" de un programa son los datos que llegan al programa desde el exterior.

Entrada por teclado: la función input()

Python proporciona funciones internas que obtienen entradas desde el teclado.
La más sencilla se llama **input**. Cuando llamamos a esta función, el programa se detiene y espera a que el usuario escriba algo. Cuando el usuario pulsa la tecla Enter, el programa se reanuda e input devuelve lo que el usuario escribió como tipo String.
Antes de llamar a input es conveniente mostrar un mensaje que le pida al usuario el dato solicitado. Este mensaje se llama indicador (prompt en inglés).

```
print("¿Cómo se llama?")
nombre = input()
print("Me alegro de conocerle,", nombre)
```

En el ejemplo anterior, el usuario escribe su respuesta en una línea distinta a la pregunta porque Python añade un salto de línea al final de cada print().
Si se prefiere que el usuario escriba su respuesta a continuación de la pregunta, se podría utilizar el argumento opcional end en la función print(), que indica el carácter o caracteres a utilizar en vez del salto de línea. Para separar la respuesta de la pregunta se ha añadido un espacio al final de la pregunta.

```
print("¿Cómo se llama? ", end="")
nombre = input()
print("Me alegro de conocerle,", nombre)
```

Otra solución, más compacta, es aprovechar que a la función input() se le puede enviar un argumento que se escribe en la pantalla (sin añadir un salto de línea):

```
nombre = input("¿Cómo se llama? ")
print("Me alegro de conocerle,", nombre)
```

Conversión de tipos

De forma predeterminada, la función input() convierte la entrada en una cadena. Si se quiere que Python interprete la entrada como un número entero, se debe utilizar la función int() de la siguiente manera:

```
cantidad = int(input("Dígame una cantidad en dólares: "))
print(cantidad, "pesos son", cantidad / 15.022, "dólares")
```

De la misma manera, para que Python interprete la entrada como un número decimal, se debe utilizar la función float() de la siguiente manera:

```
cantidad = float(input("Dígame una cantidad en euros (hasta con 2
decimales): "))
print(cantidad, "dólares son", cantidad * 15.022, "pesos")
```

Si el usuario escribe un número decimal, la función int() producirá un error:

```
edad = int(input("Dígame su edad: "))
print("Su edad son", edad, "años")

Traceback (most recent call last):
  File "ejemplo.py", line 1, in <module>
    edad = int(input("Dígame su edad: "))
ValueError: invalid literal for int() with base 10:
'15.5'
```

Pero si el usuario escribe un número entero, la función float() **no** producirá un error, aunque el número se escribirá con parte decimal (.0):

```
peso = float(input("Dígame su peso en kg: "))
print("Su peso es", peso, "kg")
```

```
Dígame su peso en kg: 84
Su peso es 84.0 kg
```

La función input() sólo puede tener un argumento. Esto puede causar problemas como en el ejemplo siguiente, en el que se quiere mostrar la respuesta de la primera instrucción en la segunda pregunta:

```
nombre = input("Dígame su nombre: ")
apellido = input("Dígame su apellido, ", nombre, ": ")
print("Me alegro de conocerle,", nombre, apellido)
```

Salida:

```
Dígame su nombre: Pepito
Traceback (most recent call last):
  File "ejemplo.py", line 2, in <module>
    apellido = input("Dígame su apellido, ", nombre, ":
")
TypeError: input expected at most 1 arguments, got 3
```

Una solución consiste en separar la pregunta de la recogida de la respuesta, escribiendo dos instrucciones:

```
nombre = input("Dígame su nombre: ")
print("Dígame su apellido,", nombre, ": ", end="")
apellido = input()
print("Me alegro de conocerle,", nombre, apellido)
```

Salida:

```
Dígame su nombre: Pepito
Dígame su apellido, Pepito : Conejo
Me alegro de conocerle, Pepito Conejo
```

Estructuras de control de flujo

Las estructuras de control condicionales, son aquellas que nos permiten evaluar si una o más condiciones se cumplen, para decir qué acción vamos a realizar (ejecutar un fragmento de código u otro) dependiendo de la circunstancia. **La evaluación de condiciones**, solo **puede arrojar** 1 de 2 resultados: **verdadero o falso** (True o False).
Para describir la evaluación a realizar sobre una condición, se utilizan **operadores relacionales** (o de comparación):

Simbolo	Significado	Ejemplo
==	Igual que	5 == 7
!=	Distinto que	rojo != verde
<	Menor que	8 < 12
>	Mayor que	12 > 7
<=	Menor o igual que	12 <= 12
>=	Mayor o igual que	4 >= 5

Y para evaluar más de una condición simultáneamente, se utilizan operadores lógicos:

Operador	Ejemplo
and (y)	5 == 7 and 7 < 12
or (o)	12 == 12 or 15 < 7
xor	4 == 4 xor 9 < 3

if

La estructura de control if ... permite que un programa ejecute unas instrucciones cuando se cumplan una condición. En inglés "if" significa "si" (condición). La orden en Pyhton se escribe así:

```
if condición:
  aquí van las órdenes que se ejecutan si la condición es cierta
  y que pueden ocupar varias líneas
```

La primera línea contiene la condición a evaluar y es una expresión lógica. Esta línea debe terminar siempre por dos puntos (:).
A continuación viene el bloque de órdenes que se ejecutan cuando la condición se cumple (es decir, cuando la condición es verdadera). Es importante señalar que este bloque debe ir sangrado, puesto que Python utiliza el sangrado para reconocer las líneas que forman un bloque de instrucciones. El sangrado que se suele utilizar en Python es de cuatro espacios, pero se pueden utilizar más o menos espacios. Al escribir dos puntos (:) al final de una línea, IDLE sangrará automáticamente las líneas siguientes. Para terminar un bloque, basta con volver al principio de la línea.

Veamos un ejemplo. El programa siguiente pide un número positivo al usuario y almacena la respuesta en la variable "numero". Después comprueba si el número es negativo. Si lo es, el programa avisa que no era eso lo que se había pedido. Finalmente, el programa imprime siempre el valor introducido por el usuario.

```
numero = int(input("Escriba un número positivo: "))
if numero < 0:
    print("¡El número ingresado no es positivo!")
print("Ha escrito el número", numero)
```

Salida:

```
Escriba un número positivo: -5
¡El número ingresado no es positivo!
Ha escrito el número -5
```

If ...else ...

Vamos a ver ahora un condicional algo más complicado. ¿Qué haríamos si quisiéramos que se ejecutaran unas ciertas órdenes en el caso de que la condición no se cumpliera? Sin duda podríamos añadir otro if que tuviera como condición la negación del primero, pero el condicional tiene una segunda construcción mucho más útil.

```
print(“Ingrese Si o No")
respuesta = input("¿Desea aprender Python? ")
if respuesta == “Si”:
    print("¡Tienes un buen gusto!")
print("Gracias")

if respuesta != “Si”:
    print("Vaya, que lástima")
```

La segunda condición se puede sustituir con un else (del inglés: si no, en caso contrario). Si leemos el código vemos que tiene bastante sentido: “si respuesta es igual a S, imprime esto y esto, si no, imprime esto otro”.
La estructura de control if ... else ... permite que un programa ejecute unas instrucciones cuando se cumple una condición y otras instrucciones cuando no se cumple esa condición. En inglés "if" significa "si" (condición) y "else" significa "si no". La orden en Pyhton se escribe así:

```
if condición:
  aquí van las órdenes que se ejecutan si la condición es cierta
  y que pueden ocupar varias líneas
else:
  y aquí van las órdenes que se ejecutan si la condición es
  falsa y que también pueden ocupar varias líneas
```

La primera línea contiene la condición a evaluar. Esta línea debe terminar siempre por dos puntos (:).

A continuación viene el bloque de órdenes que se ejecutan cuando la condición se cumple (es decir, cuando la condición es verdadera). Es importante señalar que este bloque debe ir sangrado, puesto que Python utiliza el sangrado para reconocer las líneas que forman un bloque de instrucciones. El sangrado que se suele utilizar en Python es de cuatro espacios, pero se pueden utilizar más o menos espacios. Al escribir dos puntos (:) al final de una línea, IDLE sangrará automáticamente las líneas siguientes. Para terminar un bloque, basta con volver al principio de la línea.

Después viene la línea con la orden else, que indica a Python que el bloque que viene a continuación se tiene que ejecutar cuando la condición no se cumpla (es decir, cuando sea falsa). Esta línea también debe terminar siempre por dos puntos (:). La línea con la orden else no debe incluir nada más que el else y los dos puntos.

En último lugar está el bloque de instrucciones sangrado que corresponde al else.

```
print("Ingrese Si o No")
respuesta = input("¿Desea aprender Python? ")
if respuesta == "Si":
    print("¡Tienes un buen gusto!")
print("Gracias")

else:
    print("Vaya, que lástima")
```

Veamos otro ejemplo, aplicando la conversión de tipos:

```
edad = int(input("¿Cuántos años tiene? "))
if edad < 18:
    print("Es usted menor de edad")
else:
    print("Es usted mayor de edad")
print("¡Hasta la próxima!")
```

Este programa pregunta la edad al usuario y almacena la respuesta en la variable "edad". Después comprueba si la edad es inferior a 18 años. Si esta comparación es cierta, el programa escribe que es menor de edad y si es falsa escribe que es mayor de edad. Finalmente el programa siempre se despide, ya que la última instrucción está fuera de cualquier bloque y por tanto se ejecuta siempre.
Es decir, si se ingresa 17, el programa mostrará “Es usted menor de edad” y posteriormente “¡Hasta la próxima”, en cambio, si se ingresa 18 el programa mostrará “Es usted mayor de edad” y finalmente “¡Hasta la próxima!”.

Un bloque de instrucciones puede contener varias instrucciones. Todas las instrucciones del bloque tener el mismo sangrado:

```
edad = int(input("¿Cuántos años tiene? "))
if edad < 18:
    print("Es usted menor de edad")
    print("Recuerde que está en la edad de aprender")
else:
    print("Es usted mayor de edad")
    print("Recuerde que debe seguir aprendiendo")
print("¡Hasta la próxima!")
```

Se aconseja utilizar siempre el mismo número de espacios en el sangrado.
No se permite que en un mismo bloque haya instrucciones con distintos sangrados. Dependiendo del orden de los sangrados, el mensaje de error al intentar ejecutar el programa será diferente.

Sentencias condicionales anidadas

Una sentencia condicional puede contener a su vez otra sentencia anidada.

Por ejemplo, el programa siguiente "adivina" el número pensado por el usuario:

```
print("Piense un número de 1 a 4.")
print("Conteste S (sí) o N (no) a mis preguntas.")
primera = input("¿El número pensado es mayor que 2? ")
if primera == "S":
    segunda = input("¿El número pensado es mayor que 3? ")
    if segunda == "S":
        print("El número pensado es 4.")
    else:
        print("El número pensado es 3")
else:
    segunda = input("¿El número pensado es mayor que 1? ")
    if segunda == "S":
        print("El número pensado es 2.")
    else:
        print("El número pensado es 1.")
print("¡Hasta la próxima!")
```

Se pueden anidar tantas sentencias condicionales como se desee.

Más de dos alternativas: if ... elif ... else ...

La estructura de control if ... elif ... else ... permite encadenar varias condiciones. "elif" es una contracción de "else if". La orden en Pyhton se escribe así:

```
if condición_1:
  bloque 1
elif condición_2:
  bloque 2
else:
  bloque 3
```

- Si se cumple la condición 1, se ejecuta el bloque 1
- Si no se cumple la condición 1 pero sí que se cumple la condición 2, se ejecuta el bloque 2
- Si no se cumplen ni la condición 1 ni la condición 2, se ejecuta el bloque 3.

Se pueden escribir tantos bloques elif como sean necesarios. El bloque else (que es opcional) se ejecuta si no se cumple ninguna de las condiciones anteriores.

Estructuras de control iterativas

A diferencia de las estructuras de control condicionales, las iterativas (también llamadas cíclicas o bucles), nos permiten ejecutar un mismo código, de manera repetida, mientras se cumpla una condición.
En Python se dispone de dos estructuras cíclicas:

- El bucle **while**
- El bucle **for**

Las veremos en detalle a continuación.

Bucle While

En general, un bucle es una estructura de control que repite un bloque de instrucciones. Un bucle while permite repetir la ejecución de un grupo de instrucciones mientras se cumpla una condición (es decir, mientras la condición tenga el valor True) .
La sintaxis del bucle while es la siguiente:

```
while condicion:
    cuerpo del bucle
```

Cuando llega a un bucle while, Python evalúa la condición y, si es cierta, ejecuta el cuerpo del bucle. Una vez ejecutado el cuerpo del bucle, se repite el proceso (se evalúa de nuevo la condición y, si es cierta, se ejecuta de nuevo el cuerpo del bucle) una y otra vez mientras la condición sea cierta. Únicamente cuando la condición sea falsa, el cuerpo del bucle no se ejecutará y continuará la ejecución del resto del programa.
La variable o las variables que aparezcan en la condición se suelen llamar variables de control. Las variables de control deben definirse **antes** del bucle while y modificarse en el bucle while.
Por ejemplo, el siguiente programa escribe los números del 1 al 3:

```
i = 1
while i <= 3:
  print(i)
  i += 1
print("Programa terminado")
```

Salida:

```
1
2
3
Programa terminado
```

El número de iteraciones no está definida antes de empezar el bucle

Bucles infinitos

Si la condición del bucle se cumple siempre, el bucle no terminará nunca de ejecutarse y tendremos lo que se denomina un **bucle infinito**..
Los bucles infinitos no intencionados deben evitarse pues significan perder el control del programa. Para interrumpir un bucle infinito, hay que pulsar la combinación de teclas **Ctrl+C**. Al interrumpir un programa se mostrará un mensaje de error similar a éste:

```
Traceback (most recent call last):
  File "ejemplo.py", line 3, in <module>
    print(i)
KeyboardInterrupt
```

Estos algunos ejemplos de bucles infinitos:

- El programador ha olvidado modificar la variable de control dentro del bucle
- El programador ha escrito una condición que se cumplirá

Bucle for

Un bucle for es un bucle que repite el bloque de instrucciones un número prederminado de veces. El bloque de instrucciones que se repite se suele llamar cuerpo del bucle y cada repetición se suele llamar iteración.
La sintaxis de un bucle for es la siguiente:

```
for variable in elemento iterable (lista, cadena, range, etc.):
    cuerpo del bucle
```

No es necesario definir la variable de control antes del bucle (como sucedía con el while), aunque se puede utilizar como variable de control una variable ya definida en el programa.
El cuerpo del bucle se ejecuta tantas veces como elementos tenga el elemento recorrible (elementos de una lista o de un range(), caracteres de una cadena, etc.). Por ejemplo:

```
print("Comienzo")
for i in [3, 4, 5]:
    print("Hola. Ahora i vale", i, "y su cuadrado", i ** 2)
print("Final")
```

Salida

```
Comienzo
Hola. Ahora i vale 3 y su cuadrado 9
Hola. Ahora i vale 4 y su cuadrado 16
Hola. Ahora i vale 5 y su cuadrado 25
Final
```

En este ejemplo hay que tener en cuenta que la variable de control va tomando los valores del elemento que se recorre.
La lista puede contener cualquier tipo de elementos, no sólo números. El bucle se repetirá siempre tantas veces como elementos tenga la lista y la variable irá tomando los valores de uno en uno. Por ejemplo:

```
print("Comienzo")
for i in ["Alba", "Benito", 27]:
    print("Hola. Ahora i vale", i)
print("Final")
```

Salida

```
Comienzo
Hola. Ahora i vale Alba
Hola. Ahora i vale Benito
Hola. Ahora i vale 27
Final
```

La costumbre más extendida es utilizar la letra *i* como nombre de la variable de control, pero se puede utilizar cualquier otro nombre válido.

En vez de una lista se puede escribir una cadena, en cuyo caso la variable de control va tomando como valor cada uno de los caracteres:

```
for i in "AMIGO":
    print("Dame una ", i)
print("¡AMIGO!")
```

Salida

```
Dame una A
Dame una M
Dame una I
Dame una G
Dame una O
¡AMIGO!
```

Bucles anidados

Se habla de bucles anidados cuando un bucle se encuentra en el bloque de instrucciones de otro bloque.
Al bucle que se encuentra dentro del otro se le puede denominar bucle interior o bucle interno. El otro bucle sería el bucle exterior o bucle externo.
Los bucles pueden tener cualquier nivel de anidamiento (un bucle dentro de otro bucle dentro de un tercero, etc.).
Aunque en Python no es necesario, se recomienda que los nombres de las variables de control de los bucles anidados no coincidan, para evitar ambigüedades.

```
for i in [0, 1, 2]:
    for j in [0, 1]:
        print("i vale", i, "y j vale", j)
```

Salida:

```
i vale 0 y j vale 0
i vale 0 y j vale 1
i vale 1 y j vale 0
i vale 1 y j vale 1
i vale 2 y j vale 0
i vale 2 y j vale 1
```

En el ejemplo anterior, el bucle externo (el controlado por i) se ejecuta 3 veces y y el bucle interno (el controlado por j) se ejecuta dos veces por cada valor de i. Por ello la instrucción print() se ejecuta en total 6 veces (3 veces que se ejecuta el bucle externo x 2 veces que se ejecuta cada vez el bucle interno = 6 veces).
En general, el número de veces que se ejecuta el bloque de instrucciones del bucle interno es el producto de las veces que se ejecuta cada bucle.
En el ejemplo anterior se han utilizado listas para facilitar la comprensión del funcionamiento del bucle pero, si es posible hacerlo, se recomienda utilizar tipos range(), entre otros motivos porque durante la ejecución del programa ocupan menos memoria en el ordenador y se pueden hacer depender del desarrollo del programa.
El siguiente programa es equivalente al ejemplo anterior:

```
for i in range(3):
    for j in range(2):
        print("i vale", i, "y j vale", j)
```

Al escribir bucles anidados, hay que prestar atención al sangrado de las instrucciones, ya que ese sangrado indica a Python si una instrucción forma parte de un bloque u otro

Nota: La sentencia break, como en C, termina el lazo for o while más anidado.

Recorrer una lista

Se puede recorrer una lista de principio a fin de dos formas distintas:

- Una forma es recorrer directamente los elementos de la lista, es decir, que la variable de control del bucle tome los valores de la lista que estamos recorriendo:

```
letras = ["A", "B", "C"]
for i in letras:
    print(i, end=" ")
```

- La otra forma es recorrer indirectamente los elementos de la lista, es decir, que la variable de control del bucle tome como valores los índices de la lista que estamos recorriendo (0,1 ,2 , etc.). En este caso, para acceder a los valores de la lista hay que utilizar letras[i]:

```
letras = ["A", "B", "C"]
for i in range(len(letras)):
    print(letras[i], end=" ")
```

La primera forma es más sencilla, pero sólo permite recorrer la lista de principio a fin y utilizar los valores de la lista.
La segunda forma es más complicada, pero permite más flexibilidad

Recorrer y modificar los elementos de una lista

```
letras = ["A", "B", "C"]
print(letras)
for i in range(len(letras)):
    letras[i] = "X"
    print(letras)
```

Salida:

```
['A', 'B', 'C']
['X', 'B', 'C']
['X', 'X', 'C']
['X', 'X', 'X']
```

Eliminar elementos de la lista

Para eliminar los elementos de una lista necesitamos recorrer la lista al revés. Si recorremos la lista de principio a fin, al eliminar un valor de la lista, la lista se acorta y cuando intentamos acceder a los últimos valores se produce un error de índice fuera de rango.

```
letras = ["A", "B", "C"]
print(letras)
for i in range(len(letras)-1, -1, -1):
    if letras[i] == "B":
        del letras[i]
    print(letras)
```

Salida:

```
['A', 'B', 'C']
['A', 'B', 'C']
['A', 'C']
['A', 'C']
```

Saber si un valor está o no en una lista

Para saber si un valor está en una lista se puede utilizar el operador in. La sintaxis sería "elemento in lista" y devuelve un valor lógico: Truesi el elemento está en la lista, False si el elemento **no** está en la lista.
Por ejemplo, el programa siguiente comprueba si el usuario es una persona autorizada:

```
personas_autorizadas = ["Alberto", "Carmen"]
nombre = input("Dígame su nombre: ")
if nombre in personas_autorizadas:
  print("Está autorizado")
else:
  print("No está autorizado")
```

Para saber si un valor **no** está en una lista se pueden utilizar los operadores not in. La sintaxis sería "elemento not in lista" y devuelve un valor lógico: True si el elemento **no** está en la lista, False si el elemento está en la lista.
Por ejemplo, el programa siguiente comprueba si el usuario es una persona autorizada:

```
personas_autorizadas = ["Alberto", "Carmen"]
nombre = input("Dígame su nombre: ")
if nombre not in personas_autorizadas:
  print("No está autorizado")
else:
  print("Está autorizado")
```

Definiendo Funciones

En Python, la definición de funciones se realiza mediante la instrucción def más un nombre de función descriptivo -para el cuál, aplican las mismas reglas que para el nombre de las variables- seguido de paréntesis de apertura y cierre.

Las funciones se pueden crear en cualquier punto de un programa, escribiendo su definición.
La primera línea de la definición de una función contiene:

- La palabra reservada def
- El nombre de la función (la guía de estilo de Python recomienda escribir todos los caracteres en minúsculas separando las plabras por guiones bajos)
- Paréntesis (que pueden incluir los argumentos de la función, como se explica más adelante)

Las instrucciones que forman la función se escriben con sangría con respecto a la primera línea.
Por comodidad, se puede indicar el final de la función con la palabra reservada return, aunque no es obligatorio.
Para poder utilizar una función en un programa se tiene que haber definido antes. Por ello, normalmente las definiciones de las funciones se suelen escribir al principio de los programas.
El ejemplo siguiente muestra un programa que contiene una función y el resultado de la ejecución de ese programa.

```
def licencia():
    print("Copyright 2013 Bartolomé Sintes Marco")
    print("Licencia CC-BY-SA 3.0")
    return

print("Este programa no hace nada interesante.")
licencia()
print("Programa terminado.")
```

Salida:

```
Este programa no hace nada interesante.
Copyright 2013 Bartolomé Sintes Marco
Licencia CC-BY-SA 3.0
Programa terminado.
```

Variables en funciones

Si la subrutina que pegamos en un programa utiliza alguna variable auxiliar para algún cálculo intermedio y resulta que el programa ya utilizaba una variable con el mismo nombre que esa variable auxiliar, los cambios en la variable que se hagan en la subrutina podrían afectar al resto del programa de forma imprevista.
Para resolver el problema de los conflictos de nombres, los lenguajes de programación limitan lo que se llama el alcance o el ámbito de las variables. Es decir, que los lenguajes de programación permiten que una variable exista únicamente en el interior de una subrutina y no afecte a otras variables de mismo nombre situadas fuera de esa subrutina.
Aunque cada lenguaje tiene sus particularidades, el mecanismo más habitual se basa en los siguientes principios:

- Cada variable pertenece a un ámbito determinado: al programa principal o a una subrutina.
- Las variables son completamente inaccesibles en los ámbitos superiores al ámbito al que pertenecen
- Las variables pueden ser accesibles o no en ámbitos inferiores al ámbito al que pertenecen (el lenguaje puede permitir al programador elegir o no esa accesibilidad)
- En cada subrutina, las variables que se utilizan pueden ser entonces:
    - Variables locales: las que pertenecen al ámbito de la subrutina (y que pueden ser accesibles a niveles inferiores)
    - Variables libres: las que pertenecen a ámbitos superiores pero son accesibles en la subrutina

Variables locales

Si no se han declarado como globales o no locales, las variables **a las que se asigna valor** en una función se consideran variables **locales**, es decir, sólo existen en la propia función, incluso cuando en el programa exista una variable con el mismo nombre, como muestra el siguiente ejemplo:

```
def subrutina():
    a = 2
    print(a)
    return

a = 5
subrutina()
print(a)
```

Salida:

```
2
5
```

Las variables locales sólo existen en la propia función y no son accesibles desde niveles superiores, como puede verse en el siguiente ejemplo:

```
def subrutina():
    a = 2
    print(a)
    return

subrutina()
print(a)
```

Salida:

```
2
Traceback (most recent call last):
  File "ejemplo.py", line 7, in <module>
    print(a)
NameError: name 'a' is not defined
```

Variables libres globales o no locales

Si a una variable **no se le asigna valor** en una función, Python la considera **libre** y busca su valor en los niveles superiores de esa función, empezando por el inmediatamente superior y continuando hasta el programa principal. Si a la variable se le asigna valor en algún nivel intermedio la variable se considera **no local** y si se le asigna en el programa principal la variable se considera **global**, como muestra.

En el ejemplo siguiente, la variable libre "a" de la función subrutina() se considera global porque obtiene su valor del programa principal:

```
def subrutina():
    print(a)
    return

a = 5
subrutina()
print(a)
```

Salida:

```
5
5
```

Variables declaradas global o nonlocal

Si queremos asignar valor a una variable en una subrutina, pero no queremos que Python la considere local, debemos declararla en la función como global o nonlocal, como muestran los ejemplos siguientes:

- En el ejemplo siguiente la variable se declara como global, para que su valor sea el del programa principal:

```
def subrutina():
    global a
    print(a)
    a = 1
    return

a = 5
subrutina()
print(a)
```

Salida:

```
5
1
```

- En el ejemplo siguiente la variable se declara como nonlocal, para que su valor sea el de la función intermedia:

```
def subrutina():
    def sub_subrutina():
        nonlocal a
        print(a)
        a = 1
        return

    a = 3
    sub_subrutina()
    print(a)
    return

a = 4
subrutina()
print(a)
```

Salida:

```
3
1
4
```

Si a una variable declarada global o nonlocal en una función no se le asigna valor en el nivel superior correspondiente, Python dará un error de sintaxis.

Sobre los parámetros

Un parámetro es un valor que la función espera recibir cuando sea llamada (invocada), a fin de ejecutar acciones en base al mismo. Una función puede esperar uno o más parámetros (que irán separados por una coma) o ninguno.

```
def mi_funcion(nombre, apellido):
    # algoritmo
```

Los parámetros, se indican entre los paréntesis, **a modo de variables**, a fin de poder utilizarlos como tales, dentro de la misma función.

Argumentos y devolución de valores

Las funciones en Python admiten argumentos en su llamada y permiten devolver valores. Estas posiblidades permiten crear funciones más útiles y fácilmente reutilizables, es decir, permiten que se les envíen valores con los que trabajar. De esa manera, las funciones se pueden reutilizar más fácilmente, como muestra el ejemplo siguiente:

```
def calcula_media(x, y):
    resultado = (x + y) / 2
    return resultado

a = 3
b = 5
media = calcula_media(a, b)
print("La media de", a, "y", b, "es:", media)
print("Programa terminado")
```

Salida:

```
La media de 3 y 5 es: 4.0
Programa terminado
```

Esta función tiene un inconveniente y es que sólo calcula la media de dos valores. Sería más interesante que la función calculara la media de cualquier cantidad de valores.
Para evitar ese problema, las funciones pueden admitir una cantidad indeterminada de valores, como muestra el ejemplo siguiente:

```
def calcula_media(*args):
    total = 0
    for i in args:
        total += i
    resultado = total / len(args)
    return resultado

a, b, c = 3, 5, 10
media = calcula_media(a, b, c)
print("La media de", str(a)+",", b, "y", c, "es:", media)
print("Programa terminado")
```

Salida:

```
La media de 3, 5 y 10 es: 6.0
Programa terminado
```

Las funciones pueden devolver varios valores simultáneamente, como muestra el siguiente ejemplo:

```
def calcula_media_desviacion(*args):
    total = 0
    for i in args:
        total += i
    media = total / len(args)
    total = 0
    for i in args:
        total += (i - media) ** 2
    desviacion = (total / len(args)) ** 0.5
    return media, desviacion

a, b, c, d = 3, 5, 10, 12
media, desviacion_tipica = calcula_media_desviacion(a, b, c, d)
print("Datos:", a, b, c, d)
print("Media:", media)
print("Desviación típica:", desviacion_tipica)
print("Programa terminado")
```

Salida:

```
Datos: 3 5 10 12
Media: 7.5
Desviación típica: 3.640054944640259
Programa terminado
```

Conflictos entre nombres de parámetros y nombre de variables globales

En Python no se producen conflictos entre los nombres de los parámetros y los nombres de las variables globales. Es decir, el nombre de un parámetro puede coincidir o no con el de una variable global, pero Python no los confunde: en el ámbito de la función el parámetro hace siempre referencia al dato recibido y no a la variable global y los programas producen el mismo resultado.
Eso nos facilita reutilizar funciones en otros programas sin tener que preocuparnos por este detalle.

Paso por valor o paso por referencia

En los lenguajes en los que las variables son "cajas" en las que se guardan valores, cuando se envía una variable como argumento en una llamada a una función suelen existir dos posiblidades:

- Paso por valor: se envía simplemente el valor de la variable, en cuyo caso la función no puede modificar la variable, pues la función sólo conoce su valor, pero no la variable que lo almacenaba.
- Paso por referencia: se envía la dirección de memoria de la variable, en cuyo caso la función sí que puede modificar la variable.

En Python no se hace ni una cosa ni otra. En Python cuando se envía una variable como argumento en una llamada a una función lo que se envía es la referencia al objeto al que hace referencia la variable. Dependiendo de si el objeto es mutable o inmutable, la función podrá modificar o no el objeto.

Clases y Objetos

Para entender este paradigma primero tenemos que comprender qué es una clase y qué es un objeto. Un objeto es una entidad que agrupa un estado y una funcionalidad relacionadas. El estado del objeto se define a través de variables llamadas atributos, mientras que la funcionalidad se modela a través de funciones a las que se les conoce con el nombre de métodos del objeto.

Un ejemplo de objeto podría ser un auto, en el que tendríamos atributos como la marca, el número de puertas o el tipo de carburante y métodos como arrancar y parar. O bien cualquier otra combinación de atributos y métodos según lo que fuera relevante para nuestro programa.
Una clase, por otro lado, no es más que una plantilla genérica a partir de la cuál instanciar los objetos; plantilla que es la que define qué atributos y métodos tendrán los objetos de esa clase.
Volviendo a nuestro ejemplo: en el mundo real existe un conjunto de objetos a los que llamamos autos y que tienen un conjunto de atributos comunes y un comportamiento común, esto es a lo que llamamos clase. Sin embargo, mi auto no es igual que el auto de mi vecino, y aunque pertenecen a la misma clase de objetos, son objetos distintos.
En Python las clases se definen mediante la palabra clave `class` seguida del nombre de la clase, dos puntos (:) y a continuación, indentado, el cuerpo de la clase. Como en el caso de las funciones, si la primera línea del cuerpo se trata de una cadena de texto, esta será la cadena de documentación de la clase o docstring.

```
class Auto:
    “””Abstraccion de los objetos auto.”””
    def __init__(self, gasolina):
         self.gasolina = gasolina
         print “Tenemos”, gasolina, “litros”

    def arrancar(self):
         if self.gasolina > 0:
              print “Arranca”
         else:
              print “No arranca”

def conducir(self):
    if self.gasolina > 0:
         self.gasolina -= 1
         print “Quedan”, self.gasolina, “litros”
    else:
       print “No se mueve”
```

Lo primero que llama la atención en el ejemplo anterior es el nombre tan curioso que tiene el método __init__. Este nombre es una convención y no un capricho. El método __init__, con un doble guión bajo al principio y final del nombre, se ejecuta justo después de crear un nuevo objeto a partir de la clase, proceso que se conoce con el nombre de instanciación. El método __init__ sirve, como sugiere su nombre, para realizar cualquier proceso de inicialización que sea necesario.
Como vemos el primer parámetro de __init__ y del resto de métodos de la clase es siempre self. Esto sirve para referirse al objeto actual. Este mecanismo es necesario para poder acceder a los atributos y métodos del objeto diferenciando, por ejemplo, una variable local mi_var de un atributo del objeto self.mi_var.
Si volvemos al método __init__ de nuestra clase Auto veremos cómo se utiliza self para asignar al atributo gasolina del objeto (self.gasolina) el valor que el programador especificó para el parámetro gasolina. El parámetro gasolina se destruye al final de la función, mientras que el atributo gasolina se conserva (y puede ser accedido) mientras el objeto viva.
Para crear un objeto se escribiría el nombre de la clase seguido de cualquier parámetro que sea necesario entre paréntesis. Estos parámetros son los que se pasarán al método __init__, que como decíamos es el método que se llama al instanciar la clase.
mi_auto = Auto(3)
Se preguntará entonces cómo es posible que a la hora de crear nuestro primer objeto pasemos un solo parámetro a __init__, el número 3, cuando la definición de la función indica claramente que precisa de

dos parámetros (self y gasolina). Esto es así porque Python pasa el primer argumento (la referencia al objeto que se crea) automágicamente.
Ahora que ya hemos creado nuestro objeto, podemos acceder a sus atributos y métodos mediante la sintaxis objeto.atributo y objeto.metodo():

```
        >>> print mi_auto.gasolina
        3
        >>> mi_auto.arrancar()
        Arranca
        >>> mi_auto.conducir()
        Quedan 2 litros
        >>> mi_auto.conducir()
        Quedan 1 litros
        >>> mi_auto.conducir()
        Quedan 0 litros
>>> mi_auto.conducir()
        No se mueve
        >>> mi_auto.arrancar()
        No arranca
        >>> print mi_auto.gasolina
         0
```

Como último apunte recordar que en Python, como ya se comentó en repetidas ocasiones anteriormente, todo son objetos. Las cadenas, por ejemplo, tienen métodos como `upper`(), que devuelve el texto en mayúsculas o `count`(sub), que devuelve el número de veces que se encontró la cadena sub en el texto.

Excepciones

Las excepciones son errores detectados por Python durante la ejecución del programa. Cuando el intérprete se encuentra con una situación excepcional, como el intentar dividir un número entre 0 o el intentar acceder a un archivo que no existe, este genera o lanza una excepción, informando al usuario de que existe algún problema.
Si la excepción no se captura el flujo de ejecución se interrumpe y se muestra la información asociada a la excepción en la consola de forma que el programador pueda solucionar el problema.
Veamos un pequeño programa que lanzaría una excepción al intentar dividir 1 entre 0.

```
def division(a, b):
return a / b
def calcular():
division(1, 0)
calcular()
```

Si lo ejecutamos obtendremos el siguiente mensaje de error:

```
$ python ejemplo.py
Traceback (most recent call last):
File “ejemplo.py”, line 7, in
calcular()
File “ejemplo.py”, line 5, in calcular
division(1, 0)
File “ejemplo.py”, line 2, in division
a / b
ZeroDivisionError: integer division or modulo by zero
```

Lo primero que se muestra es el trazado de pila o *traceback*, que consiste en una lista con las llamadas que provocaron la excepción. Como vemos en el trazado de pila, el error estuvo causado por la llamada a calcular() de la línea 7, que a su vez llama a division(1, 0) en la línea 5 y en última instancia por la ejecución de la sentencia a / b de la línea 2 de division.
A continuación vemos el tipo de la excepción, ZeroDivisionError, junto a una descripción del error: “integer division or modulo by zero” (módulo o división entera entre cero).
En Python se utiliza una construcción try-except para capturar y tratar las excepciones. El bloque try (intentar) define el fragmento de código en el que creemos que podría producirse una excepción. El bloque except (excepción) permite indicar el tratamiento que se llevará a cabo de producirse dicha excepción. Muchas veces nuestro tratamiento de la excepción consistirá simplemente en imprimir un mensaje más amigable para el usuario, otras veces nos interesará registrar los errores y de vez en cuando podremos establecer una estrategia de resolución del problema.
En el siguiente ejemplo intentamos crear un objeto f de tipo fichero. De no existir el archivo pasado como parámetro, se lanza una excepción de tipo IOError, que capturamos gracias a nuestro try-except.

```
try:
    f = file(“archivo.txt”)
except:
    print “El archivo no existe”
```

Python permite utilizar varios except para un solo bloque try, de forma que podamos dar un tratamiento distinto a la excepción dependiendo del tipo de excepción de la que se trate. Esto es una buena práctica, y es tan sencillo como indicar el nombre del tipo a continuación del except.

```
try:
    num = int(“3a”)
    print no_existe
except NameError:
    print “La variable no existe”
except ValueError:
    print “El valor no es un numero”
```

Cuando se lanza una excepción en el bloque try, se busca en cada una de las clausulas except un manejador adecuado para el tipo de error que se produjo. En caso de que no se encuentre, se propaga la excepción.
Además podemos hacer que un mismo except sirva para tratar más de una excepción usando una tupla para listar los tipos de error que queremos que trate el bloque:

```
try:
    num = int("3a")
    print no_existe
except (NameError, ValueError):
    print "Ocurrio un error"
```

La construcción try-except puede contar además con una clausula else, que define un fragmento de código a ejecutar sólo si no se ha producido ninguna excepción en el try.

```
try:
    num = 33
except:
    print "Hubo un error!"
else:
    print "Todo esta bien"
```

También existe una clausula finally que se ejecuta siempre, se produzca o no una excepción. Esta cláusula se suele utilizar, entre otras cosas, para tareas de limpieza.

```
try:
    z = x / y
except ZeroDivisionError:
    print "Division por cero"
finally:
    print "Limpiando"
```

También es interesante comentar que como programadores podemos crear y lanzar nuestras propias excepciones. Basta crear una clase que herede de Exception o cualquiera de sus hijas y lanzarla con raise.

```
class MiError(Exception):
  def __init__(self, valor):
    self.valor = valor

  def __str__(self):
    return "Error " + str(self.valor)
try:
    if resultado > 20:
       raise MiError(33)
except MiError, e:
   print e
```

Anexo: Operaciones adicionales

Listas

`L.append(object)`: Añade un objeto al final de la lista.
`L.count(value)`: Devuelve el número de veces que se encontró value en la lista.
`L.extend(iterable)`: Añade los elementos del iterable a la lista.
`L.index(value[, start[, stop]])`: Devuelve la posición en la que se encontró la primera ocurrencia de value. Si se especifican, start y stop definen las posiciones de inicio y fin de una sublista en la que buscar.
`L.insert(index, object)`: Inserta el objeto object en la posición index.
`L.pop([index])`: Devuelve el valor en la posición index y lo elimina de la lista. Si no se especifica la posición, se utiliza el último elemento de la lista.
`L.remove(value)`: Eliminar la primera ocurrencia de value en la lista.
`L.reverse()`: Invierte la lista. Esta función trabaja sobre la propia

Cadenas

`S.count(sub[, start[, end]])`: Devuelve el número de veces que se encuentra sub en la cadena. Los parámetros opcionales start y end definen una subcadena en la que buscar.
`S.find(sub[, start[, end]])`: Devuelve la posición en la que se encontró por primera vez sub en la cadena o -1 si no se encontró.
`S.join(sequence)`: Devuelve una cadena resultante de concatenar las cadenas de la secuencia seq separadas por la cadena sobre la que se llama el método.
`S.partition(sep)`: Busca el separador sep en la cadena y devuelve una tupla con la subcadena hasta dicho separador, el separador en si, y la subcadena del separador hasta el final de la cadena. Si no se encuentra el separador, la tupla contendrá la cadena en si y dos cadenas vacías.
`S.replace(old, new[, count])`: Devuelve una cadena en la que se han reemplazado todas las ocurrencias de la cadena old por la cadena new. Si se especifica el parámetro count, este indica el número máximo de ocurrencias a reemplazar.
`S.split([sep [,maxsplit]])`: Devuelve una lista conteniendo las subcadenas en las que se divide nuestra cadena al dividirlas por el delimitador sep. En el caso de que no se especifique sep, se usan espacios. Si se especifica maxsplit, este indica el número máximo de particiones a realizar.

Bibliografía

El Tutorial de Python
Guido van Rossum
Traducido y empaquetado por la comunidad de Python Argentina

Python para Principiantes
Eugenia Bahit

Introducción a la programación con Python
Bartolomé Sintes Marco
http://www.mclibre.org/

Aprenda a Pensar Como un Programador con Python
Allen Downey, Jeffrey Elkner, Chris Meyers
Green Tea Press

Python para todos
Raúl González Duque
http://mundogeek.net/